Bolt Beranek and Newman Inc.

(12)

Report No. 5203

Development of a Voice Funnel System

Quarterly Technical Report No. 16
1 May 1982 to 31 July 1982

October 1982

Prepared for:
Defense Advanced Research Projects Agency

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A121272	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Development of a Voice Funnel System Quarterly Technical Report No. 16		5. TYPE OF REPORT & PERIOD COVERED Quarterly Technical 1 May - 31 July 1982
		6. PERFORMING ORG. REPORT NUMBER 5203
7. AUTHOR(s) J. Goodhue, Jr.		8. CONTRACT OR GRANT NUMBER(s) MDA903-78-C-0356
9. PERFORMING ORGANIZATION NAME AND ADDRESS Bolt Beranek and Newman Inc. 10 Moulton Street Cambridge, MA 02238		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS DARPA 1400 Wilson Boulevard Arlington, VA 22209		12. REPORT DATE October 1982
		13. NUMBER OF PAGES 28
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Voice Funnel, Digitized Speech, Packet Switching, Butterfly Switch, Multiprocessor.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This Quarterly Technical Report covers work performed during the period noted on the development of a high-speed interface, called a Voice Funnel, between digitizied speech streams and a packet-switching communications network.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DEVELOPMENT OF A VOICE FUNNEL SYSTEM

QUARTERLY TECHNICAL REPORT NO. 16
1 May 1982 to 31 July 1982

October 1982

This research was sponsored by the
Defense Advanced Research Projects
Agency under ARPA Order No.: 3653
Contract No.: MDA903-78-C-0356
Monitored by DARPA/IPTO
Effective date of contract: 1 September 1978
Contract expiration date: 31 December 1982
Principal investigator: R. D. Rettberg

Prepared for:

Dr. Robert E. Kahn, Director
Defense Advanced Research Projects Agency
Information Processing Techniques Office
1400 Wilson Boulevard
Arlington, VA 22209

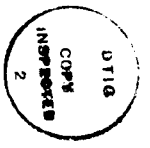

Table of Contents

FIGURES

TABLES

1. Introduction

This Quarterly Technical Report, Number 16, describes aspects of our work performed under Contract No. MDA903-78-C-0356 during the period from 1 May 1982 to 31 July 1982. This is the sixteenth in a series of Quarterly Technical Reports on the design of a packet speech concentrator, the Voice Funnel.

This report describes the set of utilities that we have developed for creating and manipulating buffer space in the Voice Funnel. In the remainder of this section, we give the motivation for the development of special-purpose facilities for Buffer Management. In Section 2, we give a general description of the Buffer Management System. In Section 3, we give detailed descriptions of the macros and system calls that are available to users.

In a communications processor such as the Voice Funnel, the mechanisms which manage buffer space are extremely important. Performance measurement experiments with the ARPANET IMP, the Pluribus Satellite IMP, and other communications processors developed at BBN have shown that the efficiency of buffer management operations has a significant impact on overall performance. As a result, we have devoted considerable attention in the development of a highly efficient set of buffer management primitives. Since the buffer management system is heavily used by the synchronous I/O software in the Voice Funnel, its features

have also been designed to match the structure of the Butterfly Synchronous I/O system.

A natural alternative would have been to use the Chrysalis Object Management System as the mechanism for buffer management, since our original intent was to use the Object Management System as a general-purpose space allocator, and we would not have needed to invest the effort in developing and documenting a second mechanism. There are two aspects of the current Object Management System that kept us from taking this course. First, the operation of mapping in an Object is relatively time consuming, as it involves checking the access privileges of the process, checking the validity of the Object Handle, and locating and setting up a Segment Attribute Register. This amount of overhead seems excessive for a Buffer Management System where the mapping in of buffers is a very frequent operation.

The second difficulty with using the Object Management System for Buffer Management stems from the fact that the Object Management System uses the memory management hardware to enforce protection between memory objects, limiting the number of legal object sizes. In particular, the smallest unit of size is 256 bytes. For many purposes, the advantages of hardware protection easily outweigh any space inefficiency due to breakage. However, in applications such as the Voice Funnel, performance is dependent to some extent on the availability of a relatively large number of buffers whose sizes do not necessarily match

hardware protection boundaries. As a result, the amount of memory space lost to due breakage could be significant in some cases if we were to use the Object Management System directly for Buffer Management.

We have taken two steps to deal with these problems. For the short term, we have developed a Buffer Management System that performs two functions: it uses the Object Management System to acquire a fixed allocation of memory space, and it supplies a set of primitive operations for manipulating buffers that are suballocated from this space. This lets user processes map in an entire buffer pool once, and use a faster (but less sophisticated) set of primitives for accessing and manipulating the buffers in that pool. It also minimizes the amount of space lost due to breakage. Unfortunately, this approach gives up many of the protection features and debugging aids offered by the Object Management System.

For the longer term, we will attempt to remedy the problems that led us to develop a special-purpose mechanism in the first place. The major thrust of this effort will be to increase the speed of the Object mapping and unmapping operations, probably through additional microcode support. The breakage problem cannot be remedied so easily, as the lower bound on the granularity of the protection sizes (256 bytes) is built into the hardware. On the other hand, memory is becoming a less expensive and more plentiful resource, since the new version of the

Butterfly Processor Node holds 256 kilobytes, expandable to four megabytes. Under these circumstances, the memory loss due to breakage may not be significant compared to the effort required to maintain a special-purpose mechanism.

2. Buffer Management

Consistent with the philosophy that all significant work on the Butterfly should be done by processes accessing local memory, the orientation of this buffer management scheme is not system-wide, but instead is directed towards individual Processor Nodes. A buffer pool is created in three steps. First, a block of memory is acquired from the Object Management System in the form of a Buffer Pool Object. This block is then broken into individual buffers. Finally, an identifier (buffer ID) is created for each buffer, and all of the buffer IDs are placed on a Dual Queue (hereafter referred to as the Free Queue). To simplify Buffer Pool deletion, ownership of the Buffer Pool Object is transferred from the creating process to the Free Queue. Any process that knows the Object Handle of the Free Queue can acquire a buffer by executing the appropriate dequeue, poll, or wait operation. There is no restriction on what object the Free Queue belongs to, or where the Free Queue is located.

A buffer pool relies on three kinds of data structure: the Free Queue, the Buffer Pool Object, and the buffer. The Free Queue is a Dual Queue as defined by the Chrysalis operating system, and is not described here. The Buffer Pool Object consists of an Object Attribute Block (OAB), which resides in Segment F8, and a memory area which resides in user memory. The type code in the OAB is the same as that of a general-purpose user object, since the treatment of these two kinds of object by

the Object Management system is for the most part identical. To differentiate a Buffer Pool Object from a general-purpose user object, a flag is set in the flags field of the OAB. The subtype field of the OAB is set to the value of the unique identifier associated with the Buffer Pool Object.

The memory area has two components. A beginning section contains the Object Handle of the Free Queue, the number and size of buffers in the pool, and space for a Dual Queue Lock. The remainder of the memory area contains the buffers themselves. As with all Chrysalis objects that incorporate user memory, the size of the buffer pool must match a hardware protection boundary size and the entire pool cannot be larger than 64 kilobytes. There is no hardware-enforced protection between buffers. If some process decides to make an access outside of a buffer that it has just acquired, it will not be prevented from doing so by the memory management hardware, unless the access is outside the Buffer Pool Object. This loss of protection is traded for smaller mapping overhead and the ability to define buffers of arbitrary size without fragmentation problems.

Buffers are not objects. They are simply contiguous blocks of memory that are suballocated from Buffer Pool Objects. A buffer consists of a header and a data area. The C structure definition of a buffer header is shown in Figure 1. A description of each of the header fields is given in Table 1. Some of these fields are used in association with Channel Control

Blocks (CCBs). CCBs are used by the synchronous I/O system, and are described in QTR 10.

```
struct buffer
{
    BUFID            buf_nxtpkt;
    BUFID            buf_next;
    short  unsigned  buf_flags;
    long   unsigned  buf_id;
    OID              buf_poolid;
    QH               buf_freeQ;
    QH               buf_lock;
    short unsigned   buf_nbytes;
    short unsigned   buf_maxsize;
    short            buf_usecnt;
    long             buf_time;
    short unsigned   buf_offset;
};
```

Figure 1 . Buffer Header

buf_nxtpkt: This field is used to construct and follow linked lists of packets.

buf_next: This field is used to construct and follow linked lists of buffers.

buf_flags: This field is reserved for status flags. When the buffer is taken from the synchronous receiver, the high order byte of the ccb_status field of the Channel Control Block is copied into the high order byte of this field by the I/O driver process. No other flags are defined.

buf_id: This field gives the identifier of the buffer. It is set on initialization and should not be changed after that.

buf_poolid: This field gives the Object Handle of the buffer pool that the buffer belongs to. It is set on initialization and should not be changed after that.

buf_freeQ: This is the Handle of the Free Queue onto which the buffer ID should be placed when the buffer is freed. The value of this field is set on initialization and should not be changed after that.

buf_lock: This field holds the handle of a Dual Queue Lock that is used to regulate access to a buffer when more than one process is manipulating its contents.

buf_nbytes: This field gives the number of bytes of valid data currently in the buffer. When the buffer is taken from the synchronous receiver, the ccb_nbytes field of the Channel Control Block is copied into this field by the I/O driver process. When a buffer is filled from scratch or modified by an application process, this field must be updated by that process.

Table 1. Buffer Fields

buf_maxsize:	This field gives the total number of bytes allocated to the buffer. It is made available for consistency checking. The value of this field is set on initialization and should not be changed after that.
buf_usecnt:	This is a use count field, available to applications where there are multiple pointers into a single buffer. Its function is described below in greater detail.
buf_time:	Much like the timestamp field of the Channel Control Blocks used in the synchronous I/O system, this field serves a dual purpose. When a buffer is taken from the synchronous receiver, the ccb_time field of the associated Channel Control Block is copied into this field by the I/O driver process. When a buffer is being prepared for output, this field holds the time at which the buffer should be transmitted. In both cases, the time specified in this field is relative to the Processor Node real time clock.
buf_offset:	This field gives the offset from the beginning of the buffer header at which the first byte of useful data resides. Its purpose is to facilitate the insertion and deletion of header fields in packet buffers. This is necessary when the Butterfly must move packets between two dissimilar networks, as in the Voice Funnel application. When the buffer is to be processed by the synchronous receiver, the "ccb_phys" field of the Channel Control Block is set by the I/O driver process to be consistent with this field. On output, the "ccb_phys" field is set to be consistent with this field when the CCB parameters are set. This field is set to "sizeof (struct buffer)" on initialization.

Table 2. Buffer Fields (continued)

Some care has been taken to ensure that operations on buffers are as efficient as possible without circumventing the protection mechanisms of the hardware and the operating system. The most important operations are gaining access to the data in a buffer and moving buffers on and off of Free Queues and other Dual Queues. Through the use of microcode-supported Dual Queue primitives, the implementation of "C" language routines that acquire and free buffers in a few tens of microseconds is not difficult.

The problem of gaining access to a buffer by using its assigned identifier is not quite as straightforward. Ideally, one would like a buffer identifier to be a logical pointer to the buffer. However, this is not practical in an environment like the Butterfly, where each process has its own segmented address space. An alternative is to use the physical address of a buffer as its identifier and allow processes to construct virtual pointers as needed. However, the construction of a virtual pointer from a physical address is only slightly less time consuming than mapping in an object; in addition, the use of physical addresses would mean giving up all the benefits of the memory protection system.

The solution adopted here is to have each process select the buffer pools it is interested in, map them in once, and maintain a table that allows buffer identifiers to be converted to virtual addresses in a small number of instructions. To accomplish this,

every buffer pool is assigned a unique (system-wide) identifier when it is created. When a process maps in a buffer pool, it stores the resulting logical pointer in a table at an offset equal to the value of the buffer pool identifier. The structure of a buffer ID is shown in Figure 2. It is the concatenation of a buffer pool identifier and the sixteen-bit offset from the buffer pool pointer at which the buffer resides. To gain access to a buffer, a process uses the buffer pool identifier to retrieve a pointer from a table of buffer pool pointers, then adds it to the offset field of the buffer (each process maintains its own table). It would be possible to eliminate the addition by creating a unique identifier for every buffer in the system, but the cost in time of the extra store and add is outweighed by the cost in space of maintaining a large table in the address space of every process in the system. With the scheme used here, the execution time of the sequence of operation needed to map in a buffer is approximately 20 microseconds on an 8 MHz MC68000. The actual time depends on whether the Buffer Identifier and table pointer are in registers or main memory.

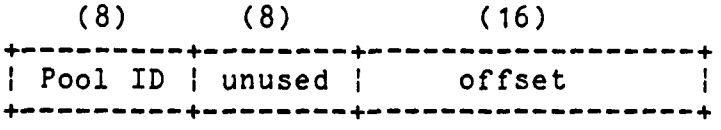

Figure 2 . Buffer Identifier

For this scheme to operate correctly, there must be a source of unique buffer pool identifiers. For this purpose, a Dual Queue is kept in a global memory segment that is shared among various operating system routines. On system initialization, the queue is filled with all legal buffer pool identifiers. When a Buffer Pool Object is created, the initialization routine dequeues the next available identifier from this queue. When a Buffer Pool Object is deleted, its identifier is placed back on the queue for reuse. A compile-time operating system constant sets the maximum number of identifiers.

Buffer pools that span more than one Processor Node can be created by assigning a single Free Queue to more than one buffer pool. This mechanism has the same generality as a mechanism that would allow a single buffer pool to span more than one node, but avoids the problems of managing a data structure across more than one node. Processes using such a pool must be careful to map in all of the associated Buffer Pool Objects. Packets that span more than one buffer are chained into linked lists using the "buf_next" field of the buffer header. By convention, the identifier of the first buffer in a multi-buffer chain serves to identify the entire chain. Chains of buffers may be linked together using the "buf_nxtpkt" field of the the first buffer of each chain.

A Buffer Pool is deleted by passing the Object Handle of the Free Queue to the Object deletion routine provided in the

Chrysalis Protected Library. This routine deletes the Buffer Pool in three steps: first, it invalidates the Object Handle of the Free Queue, denying further access to the Buffer Pool. Then, if there are Event Handles on the queue, they are all posted with null pointers as data. This wakes up any process that may be waiting on the Free Queue, and informs it that the buffer pool no longer exists. Finally, the Object Management System deletes all of the Buffer Pool Objects that belong to the Free Queue, and frees their identifiers for reuse. If any buffers are still in use, the processes that hold them will have the Buffer Pool Object mapped in, preventing the Object Management System from reusing the Buffer Pool Object and causing conflicts. Once all processes have unmapped the object, it will be returned to free storage by the Object Management System.

It is sometimes useful to give multiple processes access to a single buffer. For instance, a process that is running a reliable protocol may want to retain a pointer to an outgoing buffer which it can queue for retransmission if no acknowledgment is received. For this purpose, the Buffer Management System supports a mechanism similar to the "use count" mechanism that was developed for the Pluribus. When a process first acquires a free buffer, it sets the use count field in the header of that buffer to one, indicating that only one process currently has access to it.

When the acquiring process is finished with the buffer, it has three choices. In the simplest case, the buffer is freed. After observing that the use count is one, the system-supplied subroutine for freeing buffers decrements the use count and places the Buffer ID on its Free Queue. Alternatively, the acquiring process may want to relinquish control over the buffer and pass its identifier on to another process. In this case, the use count is left untouched because the number of processes with access to the buffer remains constant. Finally, the acquiring process may want to pass the buffer on to another process but retain its own pointer to the buffer as well. In this case, the acquiring process increments the use count of the buffer before passing its identifier on. When the system-supplied subroutine for freeing buffers encounters a buffer whose use count is greater than one, it decrements the use count but does not put the buffer identifier on the Free Queue.

When more than one process has access to a given buffer, a locking mechanism is necessary to ensure consistency. In particular, it is necessary to lock a buffer before incrementing or decrementing a use count that is greater than one. To meet this need, every buffer pool object incorporates a Dual Queue lock, and the system subroutine for freeing buffers always waits on this lock before attempting to decrement a use count that is greater than one (note that the lock need not be used when the use count is equal to one). Since a Dual Queue lock consumes

memory space of its own, this mechanism uses one lock per buffer pool, rather than one lock per buffer. As a result, there will be competition for the lock between unrelated processes under some circumstances. Since the lock is not a spin lock and it should never be held for more than a few instruction times, the time lost due to unwarranted waits on the lock should be outweighed by the amount of space that would be consumed if a separate lock were to be implemented for every buffer. The number of processes allowed to wait on the lock at any one time is set by an operating system constant.

By convention, the use count of the first buffer in a linked list of buffers is taken as the use count of the entire chain. This reduces the overhead associated with manipulating linked lists of buffers.

It is up to the processes that drive the synchronous I/O channels to take care of the special case where buffers are to be passed to or from the synchronous I/O hardware. When the ID of a buffer that belongs on a synchronous receiver queue is placed on its Free Queue, the Synchronous I/O Driver must dequeue the buffer ID, associate a CCB with the buffer, and splice the CCB onto the appropriate synchronous receiver queue. Similarly, when a buffer is freed by the Synchronous Transmitter it must be picked up by the Synchronous I/O Driver and its ID must be placed on the appropriate Free Queue.

3. Buffer Management Utility Routines

This section gives detailed specifications for the utility routines that have been developed to support the Buffer Management facilities in the Voice Funnel. For convenience, "struct buffer" is defined to be equivalent to "BUFFER". In order to minimize execution time, several of the facilities that manipulate buffers directly are implemented as C macros. The names and functions of the buffer management utilities are as follows:

Operations on Buffer pools:

Make_BFpool	- Create a buffer pool
BFmap_pool	- Map in a buffer pool
BFunmap_pool	- Unmap a buffer pool

Operations on buffers:

BFmap_buf	- Map in buffer
BFfree_buf	- Free a buffer
BFfree_chain	- Free a chain of buffers
BF_LOCK	- Lock a buffer
BF_UNLOCK	- Unlock a buffer
BFinc_use	- Indivisibly increment a use count
BFmod_offset	- Modify the "buf_offset" field

Data Transfer:

BFcopy	- Copy data from one buffer to another

Title: Make_BFpool

Function: Create a Buffer Pool Object

Arguments:

1. short number of buffers in the pool
2. QH Object Handle of Free Queue
3. short size of each buffer
4. int Processor Node on which to create object (-1 => local)
5. bits Desired protection code, (0 => use default)

Return Value: OID -- Object Handle of the buffer pool

Possible Exceptions: none

Files: /usr/butterfly/chrys/prot/buffer.c68

Description:

This routine first creates a Buffer Pool Object on the specified Processor Node and acquires an identifier for it. It then initializes the buffers in the Object, places the identifier of each buffer on the Free Queue, and transfers ownership of the Object to the Free Queue. If -1 is supplied as the Processor Node number, the buffer pool object will be created on the same node as the creating process. The buffer pool identifier is stored in the subtype field of the Object Attribute Block, and the Buffer Pool Object flag (80 hex) in the flags field of the Object Attribute Block is set in order to differentiate this object from an ordinary object with memory.

Bugs: no known bugs

Example:

```
short nbuf;
QH freeQ;
short bufsize;
short node;
bits prot;
OID pool;

pool = Make_BFpool (nbuf, freeQ, bufsize, node, prot);
```

Title: BFmap_pool

Function: map in a buffer pool

Arguments:

1. OID Buffer Pool Object Handle

Return Value: pointer to the buffer pool

Possible Exceptions:

CONSISTENCY already mapped in

Files: /usr/butterfly/lib/csrc/buffer.c68

Description:

This routine is for processes that wish to gain access to buffers in a given Buffer Pool Object. It first uses the Object Management System to map in the specified Buffer Pool Object. Then a pointer to the buffer pool is entered into a table of buffer pool pointers at an offset corresponding to the value of the Buffer Pool identifier. The table is automatically declared in the header file "buffer.h".

Bugs: no known bugs

Example:

```
OID bufpool;

BFmap_pool (bufpool);
```

Title: BFunmap_pool

Function: unmap a buffer pool

Arguments:

1. OID Buffer Pool Object Handle

Return Value: none

Possible Exceptions:

CONSISTENCY Not mapped in

Files: /usr/butterfly/lib/csrc/buffer.c68

Description:

This routine is for processes that no longer need access to the buffers in a given Buffer Pool Object. It uses the Object Management System to unmap the specified Buffer Pool Object, and removes the pointer to the buffer pool object from the global table of buffer pool pointers associated with the process.

Bugs: no known bugs

Example:

```
OID bufpool

BFunmap_pool (bufpool);
```

Title: BFmap_buf

Function: map in a buffer

Arguments:

1. BUFID Buffer Identifier

Return Value: Buffer Pointer

Possible Exceptions: none

Files: /usr/butterfly/lib/csrc/buffer.h

Description:

This macro converts a Buffer Identifier into the logical address of a buffer. To construct the pointer, it uses the global table of Buffer Pool pointers associated with the process to find a pointer to the appropriate buffer pool. It then adds in the offset specified in the low half of the buffer identifier, and returns the result. In the interest of speed, no error checking is done.

Bugs: no known bugs

Example:

```
BUFFER *bufp;
BUFID   bufid;

bufp = BFmap_buf (bufid);
```

Title: BFfree_buf

Function: free a buffer

Arguments:

1. BUFFER* pointer to a buffer

Return Value: none

Possible Exceptions: none

Files: /usr/butterfly/chrys/prot/buffer.c68

Description:

This routine first tests the use count field of the specified buffer. If the count equals one, it decrements the count and enqueues the identifier of the buffer to the Free Queue whose handle is stored in the buffer header. Otherwise, it waits on the lock whose handle is stored in the buffer header, decrements the use count, and rechecks the count. If the count is greater than or equal to one, the routine returns. Otherwise, some other process has decremented the use count while this one was waiting on the lock. In that case, the buffer identifier is placed on its Free Queue.

Bugs: no known bugs

Example:

```
BUFFER *bufp;

BFfree_buf (bufp);
```

Title: BFfree_chain

Function: free a chain of buffers

Arguments:

1. BUFFER* pointer to a buffer

Return Value: none

Possible Exceptions: none

Files: /usr/butterfly/chrys/prot/buffer.c68

Description:

This routine uses an algorithm similar to that of BFfree_buf to free all of the buffers in the the chain of buffers headed by the specified buffer. In order to reduce processing overhead, the buffer management system uses the convention that the first buffer in the chain holds status information pertinent to the entire chain. Thus, it is the use count in the first buffer of a chain that determines how many processes currently hold the chain. The use count in all but the first buffer in a chain should always be one.

Bugs: no known bugs

Example:

```
BUFFER *bufp;

BFfree_chain (bufp);
```

Title: BF_LOCK

Function: lock a buffer

Arguments:

1. BUFFER* pointer to a buffer

Return Value: none

Possible Exceptions: none

Files: /usr/butterfly/lib/csrc/buffer.c68

Description:

This macro locks a buffer by dequeueing from a Dual Queue lock associated with its pool. The possible time overhead due to lock contention is traded for the space reduction gained by having per pool, rather than per buffer, locks. A buffer must be locked when: (1) a process wants to increment its use count to a value greater than one; (2) the buffer is being freed and its use count is greater than one.

Bugs: no known bugs

Example:

```
BUFFER *bufp;

BF_LOCK (bufp);
```

Title: BF_UNLOCK

Function: unlock a buffer

Arguments:

1. BUFFER* pointer to a buffer

Return Value: none

Possible Exceptions: none

Files: /usr/butterfly/chrys/prot/buffer.h

Description:

This is the companion macro to "BF_LOCK". It unlocks a buffer by enqueuing the handle of the calling process to a Dual Queue lock associated with a buffer pool.

Bugs: no known bugs

Example:

```
BUFFER *bufp;

BF_UNLOCK (bufp);
```

Title: BFinc_use

Function: Indivisibly increment a use count

Arguments:

1. BUFFER* pointer to a buffer

Return Value: none

Possible Exceptions: none

Files: /usr/butterfly/chrys/prot/buffer.h

Description:

This macro locks the specified buffer and increments its use count. It is only needed when a process wishes to increment a non-zero use count. Note the absence of a similar macro for decrementing a use count. That is because a process should never decrement a use count unless it intends to free the associated buffer.

Bugs: no known bugs

Example:

```
BUFFER *bufp;

BFinc_use (bufp);
```

Title: BFmod_offset

Function: modify the buf_offset field of a buffer

Arguments:

1. BUFFER* pointer to the buffer
2. short increment

Return Value: none

Possible Exceptions: none

Files: /usr/butterfly/chrys/prot/buffer.h

Description:

This macro adds the specified increment to the "buf_offset" field of a buffer and subtracts it from the "buf_nbytes" field. This is intended merely as a convenient shorthand for a common pair of operations.

Bugs: no known bugs

Example:

```
BUFFER *bufp;
short increment;

BFmod_offset (bufp, increment);
```

Title: BFcopy

Function: Block transfer data of one buffer to another

Arguments:

1. BUFFER* pointer to the source buffer
2. BUFFER* pointer to the destination buffer

Return Value: none

Possible Exceptions:

CONSISTENCY destination buffer too small

Files: /usr/butterfly/chrys/prot/buffer.c68

Description:

This routine block transfers data in the source buffer to the destination buffer. Data is copied into the destination buffer starting at the offset specified in the "buf_offset" field of the destination buffer. No copying will take place if sufficient destination buffer space is not available. On completion of the transfer, the "buf_nbytes" field of the source buffer is copied into the corresponding field of the destination buffer. Data is copied using the block transfer operation provided by the Processor Node Controller, even if the two buffers are on the same node.

Bugs: no known bugs

Example:

```
BUFFER *srcp;
BUFFER *destp
unsigned short nbytes;

BFcopy (srcp, destp, nbytes);
```

DISTRIBUTION OF THIS REPORT

Defense Advanced Research Projects Agency
Dr. Robert E. Kahn (2)
Dr. Vinton Cerf (1)

Defense Supply Service -- Washington
Jane D. Hensley (1)

Defense Documentation Center (12)

USC/ISI
Danny Cohen
Steve Casner

MIT/Lincoln Labs
Dr. Clifford J. Weinstein (3)

SRI International
Earl Craighill (1)

Rome Air Development Center
Neil Marples - RBES (1)
Julian Gitlin - DCLD (1)

Bolt Beranek and Newman Inc.
Library
Library, Canoga Park Office (2)
S. Blumenthal
R. Bressler
R. Brooks
P. Carvey
P. Castleman
W. Edmond
G. Falk
J. Goodhue
S. Groff
E. Harriman
F. Heart
M. Hoffman
M. Kraley
A. Lake
W. Mann
W. Milliken
M. Nodine
R. Rettberg
P. Santos
G. Simpson
E. Starr
E. Wolf